Embedded Applications Journal

A PUBLICATION OF INTEL CORPORATION FOURTH QUARTER, 1994

What's Inside

A Low-cost Development Tool

Article ID# 1001

Intel's EP80960Cx Evaluation Platform offers a low-cost hardware tool for executing and debugging code. Featuring the i960® CA processor, it upgrades easily to the i960 CF processor, the newest and highest-performance member of Intel's family of 32-bit embedded microprocessors.

The board includes a high-performance interleaved DRAM subsystem that operates with two wait state (2-0-0-0) burst reads and zero wait state (0-0-0-0) posted writes. The 2-MB memory subsystem, expandable up to 16 MB, employs standard 70-ns DRAM SIMMs and runs at frequencies up to 33 MHz.

The board also features an I/O subsystem and an advanced set of peripheral devices for benchmarking and debugging application code written for i960 CA/CF embedded processors. Also, an X-Bus both accepts expansion cards and external devices, and gives direct access to the processor's bus and control signals.

The EP80960Cx includes an on-board, EPROM-based MON960 monitor with such popular features as a single-line assembler/disassembler, single-step program execution and software breakpoints. Also available is a complete code development environment with tools like Intel's iC-960 or GNU/960 compilers.

The complete package also includes a power supply, nine-pin PC /AT RS-232 serial connector and a 25-pin parallel port and cable for downloading software. Included as well are diskettes of the MON960 host software; design and library files; an example program; and the *EP80960Cx User's Manual*. The manual includes board schematics, parts list, programmable logic device equations, and step-by-step instructions on how to compile, assemble, link, download and execute the example program.

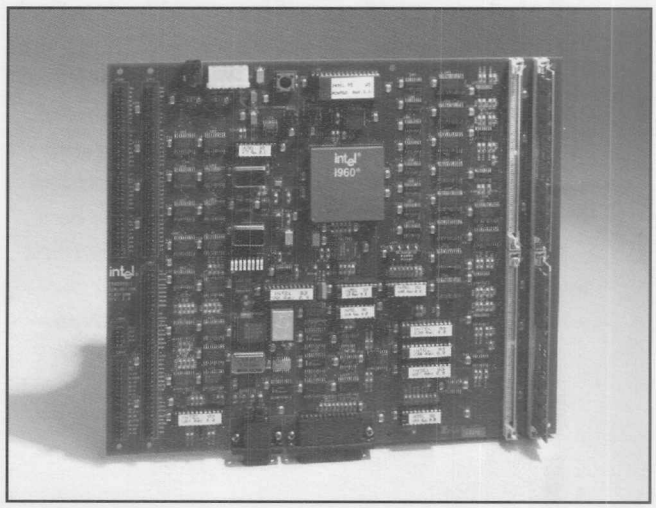


Figure 1. EP80960Cx Board

EP80960Cx Features

- 33-MHz Execution Speed
- 32-KB EPROM for 80960CA MON960 Target Operating Firmware
- Two-MB Fast Page Mode DRAM*, Expandable to 16 MB
- Concurrent Interrogation of Memory and Registers
- Centronics Port for Parallel Downloads

*A DRAM subsystem with zero wait state posted writes (0-0-0-0) and two (2-0-0-0) wait state reads.

Continued on page 9

Changing Our Way of Thinking

We recently finished several projects using fuzzy logic on Intel microcontrollers and microprocessors. You can read about our results in the *Fuzzy Logic Applications* handbook (order number 272589-001).

In addition to the encouraging results, that experience changed our approach to problem solving. All of the team members had been schooled and were well-disciplined in classic methods. We could define the parameters and write an equation or develop an algorithm to describe the activity. Newton, Galileo and Laplace, among other pioneers of the past, would have been proud.

But something curious happened along the way. We had purposely selected problems that were difficult to describe by an equation or, if they could be described, the equation was extremely difficult to solve. So here we were with inputs, a desired result and no practical equation to relate the two. At first we felt frustrated; our acquired skills provided little, it seemed, in the way of a solution. But then we realized that problem solving is problem solving, no matter how you do it.

Rather than develop a solution that was described by parameters like resistance, capacitance and mass—which would result in a precisely inexact equation—we looked at the problem as a whole. We were able to incorporate the effects of secondary, tertiary and nth-order elements. In other words, we looked at the situation rather than its parameters. To misapply Newton's Third Law, it was a case of action and reaction.

Rather than tightly couple the input to the output by means of an equation, we described the situation in broad terms and rules. The change in our thinking was startling. Rather than ask "What can it do?", the question became "What should it do?" In other words, the solution was more in tune with the problem rather than with an inexact model of the problem.

Despite its simplicity, fuzzy logic allows a much richer relationship between the inputs and the result. Consequently, it will provide new and better solutions to old problems, and solutions to problems that are presently beyond us. Fuzzy logic and its cousins, neural networks and chaos theory, offer a new set of capabilities.

Come on in, the water's fine.

Joe Altnether
Fuzzy Logic Program Director
Embedded Microcontroller Division
Intel Corporation

Intel Support Numbers

Customer Support	(U.S. and Canada)	800-628-8686
Customer Training	(U.S. and Canada)	800-234-8806
Literature Fulfillment	(U.S. and Canada)	800-468-8118
Literature Fulfillment	(Europe)	+44(0)793-431155
FaxBack* Service	(U.S. and Canada)	800-628-2283
FaxBack Service	(Europe)	+44(0)793-496646
FaxBack Service	(Worldwide)	916-356-3105
AMO Applications BBS	(Worldwide)	
up to 14.4 Kbaud lines		916-356-3600
dedicated 2400-baud lines		916-356-7209
Applications BBS	(Europe)	+44(0)793-496340

More Copies

Need more copies of this issue of the Embedded Applications Journal? Please call Intel Literature Fulfillment at 800-468-8118 in the U.S. and Canada or +44(0)793-431155 in Europe and ask for order #241294-011.

Implementing Additional External Interrupts on MCS® 51 Microcontrollers, Part 2

S.S. Dang and Y.Y. Soh
Applications Engineers
Intel Corporation
Article ID# 1002

In this second part of a two-part series, we discuss how timer 0, timer 1 and the PCA counter can serve as external interrupt inputs, supplementing dedicated external interrupt sources INT0 and INT1. Part 1, which appeared in the previous issue of Embedded Applications Journal, showed how to use timer 2 for the same purpose.

Timer 0, 1

Getting timer 0 and timer 1 to work as external interrupts is as simple as causing them to overflow and, in fact, uses that mechanism: A high-to-low transition applied to pin T0 or T1 will cause the corresponding counter, if preloaded with FFH, to overflow. The counter overflow, in turn, generates an interrupt. That interrupt points to address 0BH for timer 0 and address 01BH for timer 1.

The service routine for these timer-based interrupts must invoke the auto-reload mode and reload the counter registers with FFH. To configure timers 0 and 1 to the 8-bit auto-reload counter overflow mode, preload the counter registers THx, TLx with FFH and set the TMOD and TCON registers as follows:

For the TMOD register:

- Set timers 0 and 1 as external event counters (C/T# = 1)
- Put the counters in the 8-bit auto-reload mode (M1 = 1, M0 = 0)
- Clear the gate bit (GATE = 0)

For the TCON register:

- Set the counters running (TR0 = 1, TR1 = 1)
- Clear the counter overflow flags (TF0 = 0, TF1 = 0)
- If external interrupts 0 and 1 are being used, set external interrupt control bits and flags (IT0/1, IE0/1)

```
$INCLUDE (C51FX.REG)          ;Registers declaration
;
    ORG           00H         ;Initialize Timer 0/1 as external interrupt
    AJMP          T01_INIT    ;Timer 0/1 initialization
    ORG 0BH
    AJMP T0_ISR               ;Timer 0 interrupt vector
    ORG 01BH
    AJMP T1_ISR               ;Timer 1 interrupt vector
;
;This is the initialization routine for timers 0 and 1
;
    ORG           0100H
T01_INIT: MOV TMOD, #66H      ;Set timer 0,1 as auto-reload event counter
    MOV           TL0, #0FFH ;Preload timer 0 counter register
    MOV           TH0, #0FFH ;Timer 0 counter reload value
    MOV           TL1, #0FFH ;Preload timer 1 counter register
    MOV           TH1, #0FFH ;Timer 1 counter reload value
    MOV IP, #0AH              ;Set timer 0,1 interrupt priority
    MOV IE, #8AH              ;Enable timer 0,1 & global interrupt
    MOV TCON, #50H            ;Set timer 0,1 run bits, clear interrupt
flags
    SJMP          $           ;Wait for interrupt
;
; This is timer 0 interrupt service routine
;
T0_ISR:
;   User's routine here
;   Interrupt flag TCON.5 is cleared by hardware
    RETI
;
; This is timer 1 interrupt service routine
;
T1_ISR:
;   User's routine here
;   Interrupt flag TCON.7 is cleared by hardware
    RETI
    END
```

Figure 1. Sample Program Using the Timer 0, 1 Counter Overflow Mode to Indicate an External Interrupt

Figure 1 gives a sample program for setting the TMOD and TCON registers.

This approach precludes the use of the other features of timers 1 and 0. In addition, port pins P3.4 (or

Continued on page 4

Continued from page 3

T0) and P3.5 (or T1) must be dedicated to the external interrupts. Although the example shows both timers serving as external interrupt sources, it is also possible to configure only one of the timers this way, leaving the other to operate as a timer.

PCA

Like timers 0 and 1, the microcontroller's PCA counter can also be used as an additional external interrupt source. There are two approaches: The first uses the capture mode; the second, like the timers, uses the counter overflow mode.

```
$INCLUDE (C51FX.REG)     ;Register Declaration
;
    ORG        00H
    AJMP       PCA_INIT   ;PCA initialization
    ORG 033H
    AJMP PCA_ISR          ;PCA interrupt vector
;
;This is the PCA initialization routine
;
    ORG        0100H
PCA_INIT: MOV CMOD, #00H ;Clear all unused bits
    MOV CCAPM0, #011H     ;PCA module 0 negative edge capture
mode
    MOV        CCON, #00H ;Clear PCA run bit & interrupt flags
    MOV IP, #40H          ;Set PCA interrupt priority
    MOV IE, #0C0H         ;Enable PCA & global interrupt
    SJMP       $          ;Wait for interrupt
;
; This is the PCA interrupt service routine
;
PCA_ISR: CLR  CCON.0      ;Clear interrupt flag, CCF0
;   User's routine here
    RETI
    END
```

Figure 2. Sample Program Using PCA Module 0's Capture Mode as an Additional External Interrupt Source

Capture Mode

In the capture mode, a transition applied to pin CEXn generates an interrupt, which is vectored to the PCA interrupt vector address, 033H. The activating transition can be high-to-low, low-to-high or either by setting CCAPMn register bits CAPNn, CAPPn or both, respectively. For example, to configure PCA module 0 for a high-to-low transition, set CMOD, CCAPM0 and CCON registers as follows:

In the CMOD register:

- Clear all unused bits (CIDL, WDTE, CPS1, CPS0, ECF = 0)

In the CCAPM0 register:

- Set PCA module 0 as negative edge trigger on pin CEX0 (CAPN0 = 1)
- Enable the compare/capture interrupt (ECCF0 = 1)
- Clear the remaining bits (ECOM0, CAPP0, MAT0, TOG0 and PWM0 = 0)

In CCON register:

- Clear PCA timer run bit (CR = 0),
- Clear the capture/compare interrupt flag (CCF0 = 0) in both initialization and interrupt service rotines
- Clear the unused flags (CF, CCF4, CCF3, CCF2, CCF1 = 0)

Figure 2 shows a program that performs these functions.

In this approach, port pin P1.3 or CEX0 is dedicated to the external interrupt. The remaining modules can be used normally, but they will share the same interrupt vector with module 0. In that case, interrupts from different modules can be differentiated by checking their interrupt flags.

Counter Overflow Mode

In the counter overflow mode, a high-to-low transition applied to pin ECI will cause the PCA counter, if preloaded with FFH, to overflow. The counter overflow, in turn, will generate an interrupt that will reference address 033H, the PCA interrupt vector. To configure the PCA in the counter overflow mode, preload the counter registers CH and CL with FFH. These two registers need to be reloaded in the interrupt service routine. Do not set the CCAPMn registers, but set the CMOD and CCON registers as follows:

In the CMOD register:

- Set the PCA counter to use an external clocking source at pin ECI (CPS1, CPS0 = 1)
- Enable the counter overflow interrupt (ECF = 1)
- Clear the other bits if not used (CIDL, WDTE = 0)

In the CCON register:

- Set the counter running (CR = 1)
- Clear the counter overflow flag (CF = 0) in both initialization and interrupt routines
- Clear the remaining unused flags (CCF4, CCF3, CCF2, CCF1, CCF0 = 0)

Figure 3 shows a program that performs these functions.

Continued on page 8

Reviewing Real-Time Operating Systems

Chris Briggs
Intel Corporation
Article ID# 1003

An operating system is code that provides services to manage a computer system's resources. It separates an application from hardware by treating the hardware as an abstract machine. In its simplest form, an operating system performs only one task. Adding interrupts adds tasks in the form of the interrupt service routines. By adding interrupts and placing them into foreground and background segments, you begin to define the work of a modern operating system.

A *real-time* operating system (RTOS) is one that performs its job—managing resources—within a given time constraint to meet a system's requirements. There is no specific response time that qualifies an operating system as "real time," although certain RTOSs may specify a response time in, say, X microseconds. Rather, it is up to the target application to define what "real time" means. Within that application, failing to meet the specified response time will cause severe consequences.

A *multitasking* RTOS provides code for partitioning a design into multiple tasks, where each task runs independently of the others. Special techniques supply each task with the resources and service needed to guarantee that the application meets its time constraints. A common technique is to have a pre-emptive priority-based scheduler for the multitude of tasks.

At the heart of an RTOS are real-time kernels or executives. A kernel is the part of a multitasking system responsible for the management of tasks; that is, managing the CPU's time and communication among the tasks. Small embedded systems that cannot afford the cost or complexity of a full-featured operating system that contains a file system, I/O management and networking functions can instead use a minimal real-time pre-emptive multitasking kernel.

When an RTOS Makes Sense

Some reasons to use an RTOS are that it:

- Provides a structure for overall system design
- Provides a common programming interface to all tasks
- Organizes system design into various functional models
- Permits independent work by a group of programmers
- Reduces development time by performing simple tasks
- Helps to manage time-critical functions easily
- Simplifies debugging by decomposing system in many simple tasks
- Makes documentation easier to maintain
- Lowers long-term maintenance because of a modular design

Not surprisingly, there are also reasons *not* to use an RTOS. These are when:

- An application is very simple or has no time-critical component
- System resources cannot support a full executive
- The budget cannot bear the potential royalty costs of a commercial product

Focus on Features

As with all tools, different RTOSs have a wide range of features. Fundamentally, the RTOS you choose should let you complete your project in the shortest time. Select an RTOS that meets your needs without modification. One that doesn't will be difficult and time-consuming to modify later.

Also, don't make the mistake of selecting an RTOS simply because it is fast. If you do need the fastest response, expect to collect and compare performance data. But even then, it is more productive to put your energy into reviewing your system's timing requirements than into reviewing RTOS performance data.

In most applications, performance is the prime issue. Ironically, designers often fear that using an RTOS will actually slow their system. But as processors get faster and application programs get more complex, the relative time taken by the OS is shrinking.

In addition to performance, some features to consider when shopping for an RTOS are debugging tools, "ROM-ability," processor support, connectivity, modularity, memory-management support, priority scheduling (fixed and dynamic), kernel size, cost and royalties, the availability of source code, development software cost, POSIX compliance, disk and diskless file options, the host computer, vendor support and training.

As for memory requirements, these need not be large. Many RTOSs have small kernels at their core. And where application memory requirements continue to grow, RTOS memory needs have not. Consequently, RTOSs have decreased as a percentage of a system's total memory needs. Also, memory-management units

are moving into the embedded world. As they become more prominent, memory concerns should fade.

Functionality, cost and support are also important concerns. Functionality varies widely, ranging from a simple kernel to a full-blown operating system with file and networking features. As for cost, many kernels can be had without paying royalties. For those that do cost, the price can often be negotiated. Like functionality, cost varies widely depending on features and support. Make sure that you know what you're paying for.

Typical RTOSs run on Unix or PC hosts. Self-hosted systems can serve as a development environment and in the target environment. Historically, these have more features and functions, but less raw performance than kernels. Kernels or executives require that development be done on a separate host and operating system platform.

In the past, the highest performance applications have been handled by kernel-based systems. However, this distinction is blurring as self-hosted systems have developed "ROM-able" kernels that improve their real-time performance. At the same time, kernels are improving their development environment and functionality with, for example, I/O and file-system features.

Architecture-independent RTOSs are available on a number of target and host platforms. Companies that offer them frequently port the RTOS to as many platforms as possible. In contrast, proprietary RTOSs come from hardware-oriented firms that limit them to their platform. However, the trend has been for hardware-oriented companies to move away from proprietary software.

Other RTOS market trends are:

- As the price of faster and more sophisticated CPUs drops, the impetus is to use these processors to build more sophisticated systems. One way to extend a system's life is to leverage the use of existing code, such as an RTOS.
- As RTOSs become less expensive, many companies that sell only RTOSs will be forced to join with other tool vendors to create integrated design environments.
- RTOSs are migrating from high-end Unix plaforms to price-sensitive Windows*-based PCs. As this happens, royalty-free RTOSs will be bundled with compilers, debuggers and performance analysis tools.
- Increasingly, RTOS support is appearing for digital signal processing (DSP) and reduced instruction set computer (RISC) architectures.
- High-end systems will continue to add capabilities. Among these are performance analysis, target simulation, automated configuration utilities, board-support-generation packages and market-specific enhancements. Others include integrated configuration management, RTOS "awareness" in all tools, device driver libraries and hooks to block-level programming.
- Many RTOSs work with an integrated design environment, particularly in the 32-bit market.
- Expect to see increased vertical integration as third-party vendors (TPVs) optimize kernels. As RTOSs become similar, they will be differentiated by the vertical markets they serve, such as multimedia.
- The growing popularity of Windows is lowering the cost of tool chains.
- As memory requirements climb with the growing complexity of applications software, the relative size of RTOSs is dropping as a percentage of memory.

In Summary

This article has attempted to introduce you to real time operating systems (RTOSs). Do you need an RTOS for your application? It's up to you to decide. The reading list below should help you make that decision. If you do, remember that the RTOS industry is a mature one, and it is usually a better choice to buy from an established industry leader than from a new player. Today you can select from a number of vendors who support Intel's Embedded Architectures.

Further Reading

Real-Time Operating Systems, *EDN*, April 14,1994

Real-Time Operating Systems, *Embedded Systems Programming*, 1/92

Real-Time Buyer's Guide, *Embedded Systems Programming*, 8/93

Real-Time Atlas, *Embedded Systems Programming*, 4/91

ROMable Real-Time Kernels, *Embedded Systems Programming*, 2/90

A Portable Real-Time Kernel in C, *Embedded Systems Programming*, 5/92

Implementing a Real-Time Kernel, *Embedded Systems Programming*, 6/92

Measuring Voltages? "Look Ma, No A/D"

Joe Altnether
Fuzzy Logic Program Director
Intel Corporation
Article ID# 1004

Sometimes a circuit must measure an analog signal, but its microcontroller either lacks an A/D converter or the signal to be measured exceeds the A/D's input rating. If the system requires a fast conversion, the only choice may be to add a separate A/D circuit. But if the signal to measure is a supplemental or ancillary function or a slowly varying voltage like temperature, other microcontroller resources can be harnessed to measure the voltage.

One such A/D technique uses voltage-to-frequency conversion. The converter's pulse repetition rate, which varies with the amplitude of the input voltage, is measured by the timer/counter structure of either an MCS® 96 or an MCS 51 microcontroller. On one hand, this approach requires added components, and the time needed to measure the pulse width limits its response time. On the other hand, coupled with an optocoupler it can measure higher voltages than most A/D converters and offer electrical isolation between input and output.

Several circuits using this technique are described in the Motorola *Thyristor Data Manual*. One circuit uses a standard unijunction transistor (UJT) such as a 2N4871, which has three terminals: emitter, base 1, and base 2 (Figure 1).

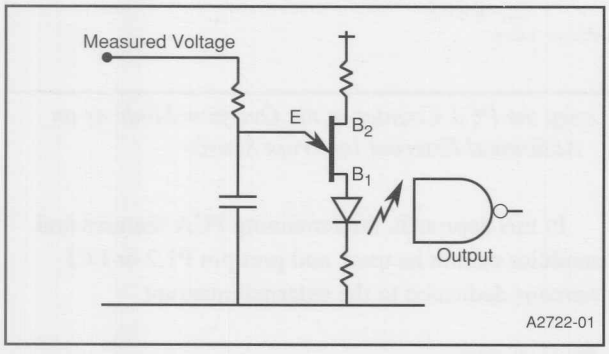


Figure 1. Circuit Using Standard Unijunction Transistor (UJT)

Between base 1 and base 2, the UJT acts like a resistor, forming a voltage divider at the emitter. The V-I curve (Figure 2) shows that only a small leakage current will flow when the emitter voltage is less than the emitter peak voltage. That's because the emitter is reverse biased. However, when the emitter voltage equals the emitter peak voltage, and the emitter current is greater than the peak point current, the device turns on, sharply dropping the base-to-base resistance.

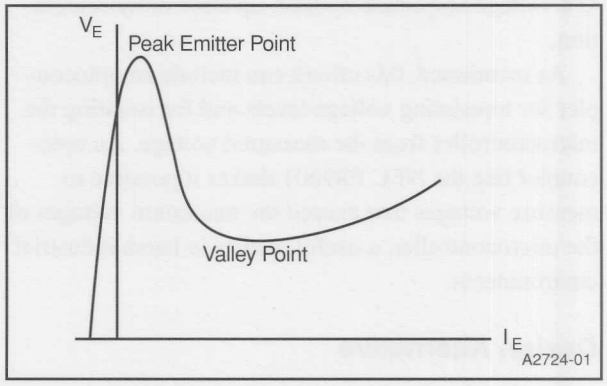


Figure 2. Emitter V-I Curve

Putting an R-C circuit on the emitter is one way to measure the time needed to reach the peak emitter, or trigger, voltage. That trigger voltage is directly proportional to V_o, the voltage across the RC network. Specifically the emitter voltage, v, is approximately equal to:

$$v = V_o e^{-t/RC}$$

where V_o is the voltage across the R-C network, R is the resistance and C is the capacitance.

Replacing V_o with the voltage we want to measure produces a trigger time that is proportional to that voltage. The actual trigger time depends on the R-C time constant and the capacitance in particular. The value of the resistor R is bounded by the operation of the UJT as follows.

$$\frac{V_s - V_p}{I_p} > R > \frac{V_s - V_v}{I_v}$$

where V_s is the supply voltage and the V_p and V_v correspond to specific peak and valley parameters of the UJT.

The higher the supply voltage, the higher the output. Typically, 20 volts is suitable. More important, however, is that the supply voltage must be greater than the expected maximum voltage to be measured.

Temperature Sensitive

Although UJTs can be temperature sensitive, carefully selecting the base-circuit resistors can compen-

sate for this effect. A good start is 15 KΩ for the resistor from the supply voltage to base 2 of the UJT and about 27Ω from base 1 to ground.

To measure the time between edges of the trigger pulses, the output of the circuit is fed into the PCA of an MCS® 51 microcontroller or the HSIO or EPA of an MCS 96 microcontroller. The information is converted to a voltage amplitude by look-up table or by calculation.

As mentioned, this circuit can include an optocoupler for translating voltage levels and for isolating the microcontroller from the measured voltage. An optocoupler like the NEC PS9601 makes it possible to measure voltages that exceed the maximum voltages of the microcontroller, a useful quality in harsh industrial environments.

Design Alternative

An alternative circuit uses a programmable unijunction transistor (PUT), which is a thryistor with a single anode (Figure 3). Regenerative feedback of the pnpn structure produces an operating region with negative resistance, making the PUT well suited as an oscillator.

With the gate voltage fixed, the device remains turned off until the anode voltage exceeds the gate voltage by one forward diode drop. Once the peak voltage is reached, the device fires. As shown, the output is isolated by a transistor at the PUT's cathode. It could also be isolated by an optocoupler.

In a fixed-frequency oscillator, a resistive voltage divider sets the bias on the gate and establishes the peak voltage, V_p.

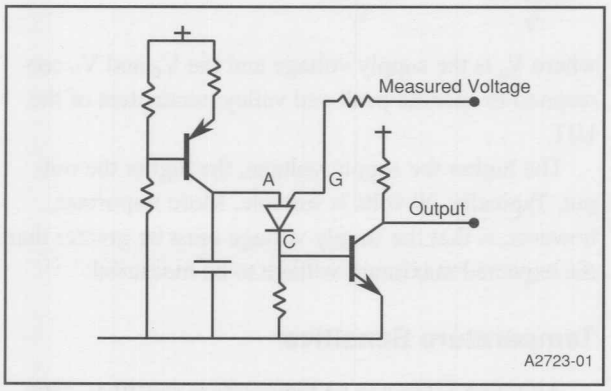


Figure 3. Circuit Using Programmable Unijunction Transistor (PUT)

The R-C time constant controls the anode voltage and, as in the previous circuit, sets the time base. A transistor serves as a constant current source, and peak voltage on the gate is modulated by the signal to be measured. When the signal to be measured is low, V_p is also low and the PUT fires faster than if V_p was higher.

With capacitance values between 0.0047 and 0.01 µF, a 15-volt span in the measured voltage will typically produce a 3- to 5-ms change in period. Higher capacitance will increase that period, widening the span between minimum and maximum voltages. The downside of adding capacitance, however, is that it also increases the measurement time.

Implementing Additional External Interrupts on MCS® 51 Microcontrollers, Part 2

Continued from page 4

```
$INCLUDE (C51FX.REG)           ;Register Declaration
;
    ORG             00H
    AJMP            PCA_INIT   ;PCA initialization
    ORG 033H
    AJMP PCA_ISR               ;PCA interrupt vector
;
;This is the PCA initialization routine
;
    ORG             0100H
PCA_INIT: MOV CMOD, #07H       ;PCA counter overflow mode
    MOV             CH, #0FFH  ;Preload PCA counter registers
    MOV             CL, #0FFH
    MOV IP, #40H               ;Set PC interrupt priority
    MOV IE, #0C0H              ;Enable PCA & global interrupt
    MOV             CCON, #40H ;Set run bit, clear interrupt flags
    SJMP            $          ;Wait for interrupt
;
; This is the PCA interrupt service routine
;
PCA_ISR: CLR        CCON.7     ;Clear interrupt flag, CF
    MOV             CH, #0FFH  ;Reload PCA counter registers
    MOV             CL, #0FFH
;   User's routine here
    RETI
    END
```

Figure 3. Using the PCA Counter in the Overflow Mode as an Additional External Interrupt Source

In this approach, the remaining PCA features and modules cannot be used, and port pin P1.2 or ECI remains dedicated to the external interrupt.

Conclusion

The above examples show that timer 0, timer 1 and the PCA can be used as external interrupt sources in addition to timer 2 (as mentioned in Part 1) and the two existing, dedicated external interrupt sources.

A Low-cost Development Tool

Continued from page 1

- X-Bus Expansion Bus for Peripheral-to-CPU Interface
- 384 Software Breakpoints
- Code Disassembler
- High-Level Language Support
- RS-232 Port for Host and User Communication at up to 115.2 Kbaud
- Memory Wait State Control for Memory Subsystem Simulation

Prototype and Expansion Bus

The X-Bus lets you prototype simple applications quickly. Two rows of header pins on the bus supply buffered address, data and control signals, as well as other signals that make it easy to attach simple peripherals and I/O devices. The header pins also serve as logic-analyzer access points.

Communication Link

Communicates with the host computer via the RS-232 link at baud rates from 300 to 115,200.

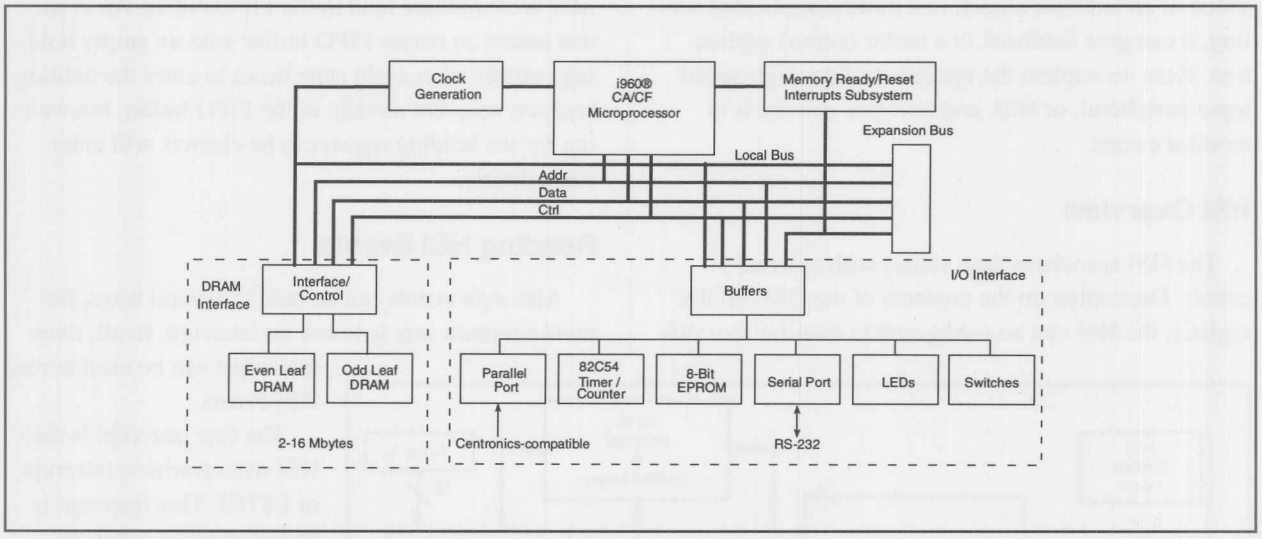


Figure 2. Block Diagram of EP80960Cx Board

Fast Page-mode DRAM SIMM Modules

The EP80960Cx Evaluation Platform's memory design takes advantage of the i960® CA/CF processor's burst mode bus for fast page-mode DRAM. For top performance, the memory subsystem combines bank interleaving, write posting and RAS interleaving.

Concurrent Reading of Memory and Registers

With the on-board MON960 monitor, you can read and modify internal registers and external memory while running your application program.

High-speed Downloads

A parallel interface serves as a full Centronics-compatible, receive-only port for quickly downloading code or data to the board.

Power Requirements

A power supply delivers the required 5 VDC at 2.5 A. The board also accepts +5 and +12 VDC for programming flash memory.

Host System Requirements

Although an Intel486-based PC is recommended, the platform connects to any Unix- or DOS-based system. A DOS-based host system must have at least:

- PC-DOS 3.2 or Later
- 512 KB of Memory
- A 1.4-MB Floppy Disk Drive
- A Serial Port (COM1 or COM2)

A parallel port (LPT1) is desirable, but not required.

Monitoring Events With the 8XC196KC/KD's High Speed Input

Sean Baartmans
Applications Support
Intel Corporation
Article ID# 1005

The high-speed input peripheral on the 8XC196KC/KD serves a simple but useful function. Used mostly to monitor events, it can be applied in many ways. In a simple application, it can measure the speed of an external object. In a more complicated setting, it can give feedback in a motor control application. Here we explain the operation of the high-speed input peripheral, or HSI, and how you can use it to monitor events.

HSI Overview

The HSI associates time values with external events. Depending on the contents of the HSI_MODE register, the HSI can be configured to monitor four different types of events: rising edges, falling edges, both rising and falling edges, or a series of eight consecutive rising edges.

When an event occurs, it is loaded into a 7 × 20-bit FIFO buffer queue (Figure 1). Eight state times later, the event is loaded into a holding register, provided that register is empty. The event can then be read from the HSI_STATUS and HSI_TIME registers. However, events are cleared from the FIFO buffer when the HSI_TIME register is read. Therefore, care must be taken to read HSI_STATUS *before* HSI_TIME, otherwise the HSI_STATUS information will be lost.

Once an event is cleared, the earliest entry in the FIFO buffer is loaded into the holding register. Depending on where the event is in the FIFO buffer, up to eight state times may be needed to load it into the holding register.

Additional events that occur before the holding register is cleared are held in the FIFO buffer. An event that enters an empty FIFO buffer with an empty holding register takes eight state times to enter the holding register; an event already in the FIFO buffer, but waiting for the holding register to be cleared, will enter immediately.

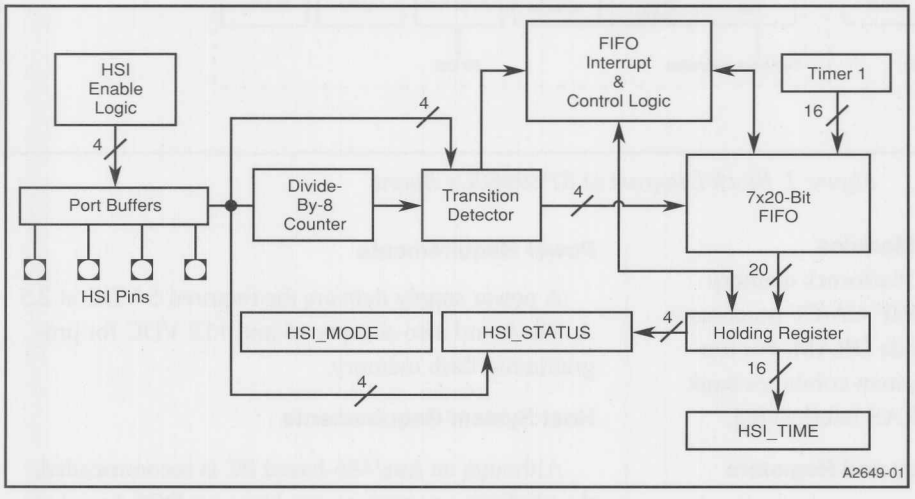


Figure 1. Block Diagram of the 8XC196KC/KD's High-Speed Input (HSI) Peripheral

Reading HSI Events

Although events can be read in several ways, the most common way is to use an interrupt. In all, three interrupts can be used to read HSI events.

The first interrupt is the HSI data-available interrupt, or INT02. This interrupt is triggered either when an event is loaded into the holding register or when the HSI FIFO buffer is full. In the first case, an interrupt pending bit is set when an event moves from the FIFO buffer to the holding register. In the second case, an interrupt pending bit is set after six entries are made into the FIFO buffer, filling the buffer. Bit IOC1.7 determines which condition will cause the interrupt: Clearing this bit selects the first case, which is an event being loaded into the holding register; setting the bit selects the second case, a filled buffer.

The second interrupt available for reading HSI events is the HSI FIFO 4 interrupt, INT10. With this interrupt you can process more than one entry at a time while minimizing the chance of the FIFO buffer becoming overloaded and, therefore, an event being missed. The interrupt pending bit is set when the FIFO

buffer loads the fourth event of the seven it can hold.

The third type of interrupt is the HSI FIFO buffer-full interrupt, INT14. It has the highest priority of the three and, in effect, is the same as FIFO 4 interrupt. The interrupt pending bit is set when the FIFO buffer is loaded with its sixth event.

Using the PTS

Once you choose the interrupt that best suits your application, you must write a service routine to read the events.

In some cases, it makes better sense to use the peripheral transaction server (PTS) to process the interrupt. The PTS is an efficient way to read events without bogging down the CPU with a long interrupt-service routine. Also, the PTS can read events from the holding register faster than a normal interrupt service routine can.

In particular, the PTS should be used when triggering an interrupt using the second or third interrupt options; that is, INT10 or INT14. That's because the PTS is tailored for transferring blocks of data, which is the correct way to handle interrupt INT10 or INT14. However, if you select the interrupt that's triggered by loading the holding register, use a standard interrupt service routine rather than the PTS.

To be serviced by the PTS, an interrupt goes to the PTS vector, which contains the location of the PTS control block. The program then jumps to the PTS control block (Table 1).

Within the control block, a variable, PTSDST, contains the address in memory that stores event data. Another variable, PTSCOUNT, decrements for each PTS cycle until reaching zero, indicating the end of a PTS cycle. The end of the cycle, in turn, generates an end-of-PTS interrupt, which points to the normal interrupt service routine vector. Thus, among its other functions, the service routine should update the PTS control block.

As an example, consider the steps needed when using the HSI data-available interrupt to monitor rising edges, make speed calculations and then transmit the result back through the serial port (Figure 3). The first step is to initialize both the serial port and the HSI unit. Using the serial port in mode 1 and HSI.0, initialization code should contain the following settings:

Table 1. PTS Control Block for HSI Mode

Unused
PTSBLOCK = 07H
Unused
Unused
PTSDST (High) = XXXXH
PTSDST (Low) = XXXXH
PTSCON = 6AH

```
WSR = 00H          ;Valid Hwindows are 0, 1 and 15
                   (selected with WSR0-3).
CCB = 0FFH         ;16-bit bus width and ready pin
                   controls wait states
BAUD_RATE = 8067H ;Bit 15 set to select internal
                    clock source;
                   ;baud rate is 9600
                   (corresponds to 8067H in mode 1)
HSI_MODE = 01H     ;HSI.0 will capture rising edges.
INT_MASK = 05H     ;Timer 1 overflow interrupt and
                   HSI data-available interrupt
                   ;enabled.
INT_MASK1 = 00H    ;No interrupts enabled in
                   interrupt mask 1
IOC0 = 01H         ;HSI.0 enabled as high-speed
input
                   ;Timer 1 overflow generates INT00,
                   P2.0 selected for serial
                   ;and transmit
IOPORT2 = 03H      ;P2.0 selected for transmit,
                   P2.1 selected for receive
SP_CON = 09H       ;Serial mode 1 selected, RXD
                   enabled
```

The next step is to write an algorithm to process the HSI routines. In this case, the routine is interrupt-driven. When an event occurs, the program jumps to the HSI data-available interrupt, calculates the time and sends the results to the serial port. In addition, because timer 1 overflow is used as the 17th bit of the timer, the HSI routine must monitor the additional count in the timer overflow routine as well as other conditions described further on. Figure 2 shows the program flow.

Table 2. 196KC/KD Interrupt Vector Sources and Locations

Interrupt Number	Interrupt Vector	Interrupt Source(s)	Interrupt Vector Location	PTS Vector Location
INT14	HSI FIFO buffer full	HSI sixth entry	203CH	205CH
INT10	HSI FIFO buffer 4	HSI FIFO buffer fourth entry	2034H	2054H
INT02	HSI data available	HSI sixth entry or HSI holding register loaded	2004H	2044H

The next step is to write the interrupt routine. Doing that successfully takes understanding the timer overflow and interrupt timing. This is especially true when using the overflow and the data-available interrupts together.

For example, when monitoring events using the HSI and timer 1, if an event occurs at FFFFH, it is possible

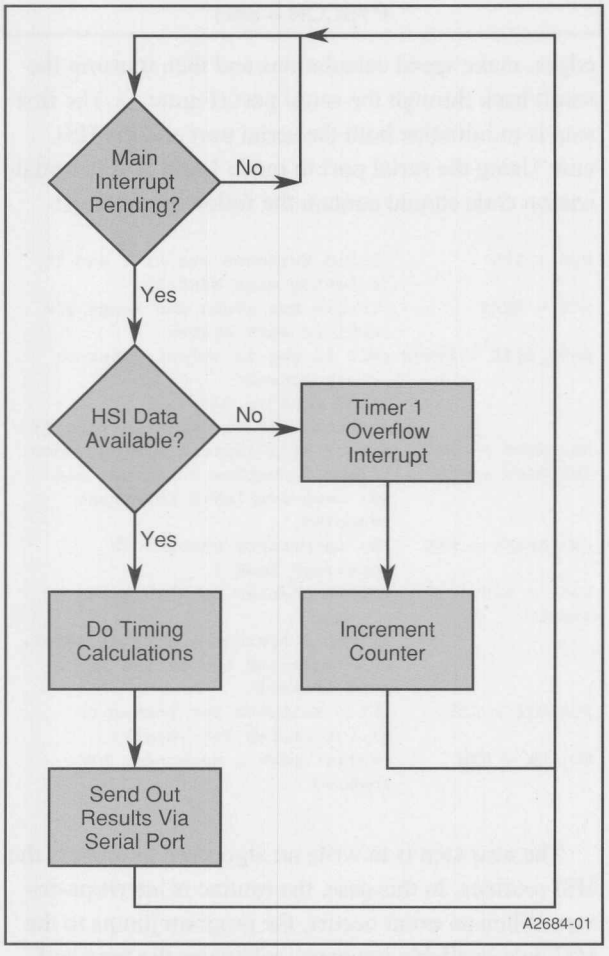


Figure 2. Example of Main Program Flow

that the timer overflow interrupt could be processed first. If that happens, and both interrupts are enabled, such a sequence would increment the timing calculation.

To understand how this happens, consider the time it takes for an HSI event to cause an interrupt: When an event triggers an HSI, it takes at least eight state times to go from the FIFO buffer to the holding register. If an event occurs on the overflow boundary, at least eight state times must pass before the corresponding interrupt-pending bit is set. The timer overflow interrupt, however, is set on the overflow boundary. If the current instruction plus the next instruction take less than eight state times, then the timer overflow interrupt occurs first.

This problem can be solved within the HSI interrupt service routine. Before jumping to an interrupt service routine, test if the HSI time equals FFFFH. If it does, then see if the timer overflow interrupt is pending. If it is, process the interrupt normally. If not, the timer overflow interrupt has occurred and needs to be accounted for. The results can be sent out via the serial port by writing to the SBUF register in horizontal window 15.

Figure 3 shows how to handle an HSI interrupt.

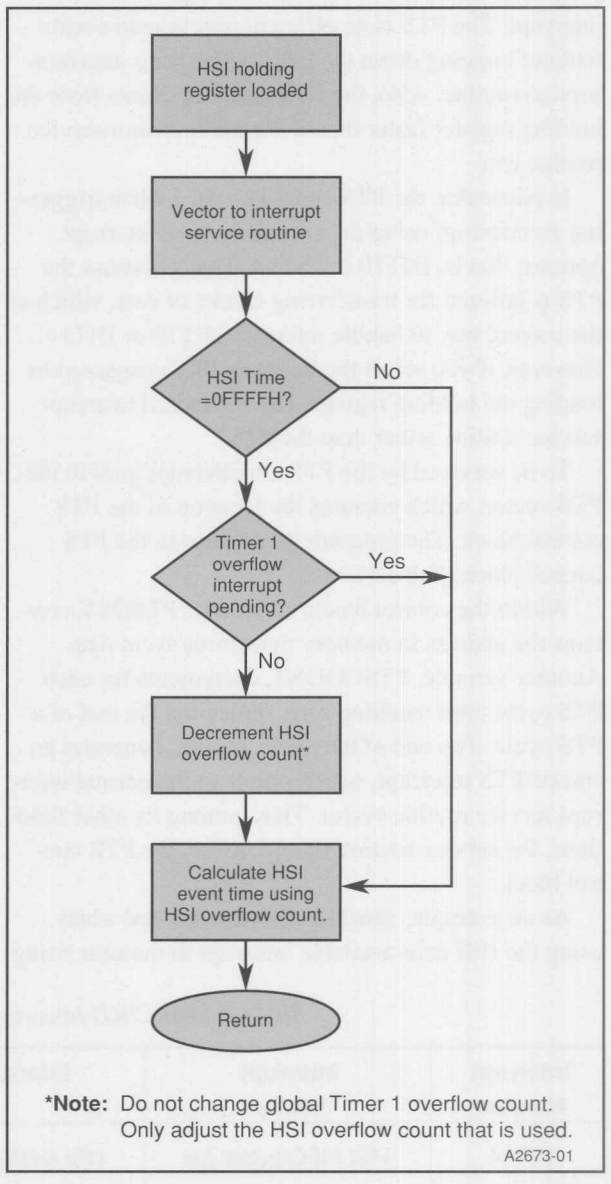


Figure 3. Sample HSI Interrupt Service Routine Block Diagram.

In summary, when using the HSI unit to monitor events:

- Decide how many HSIs you will need.
- Decide on an algorithm for reading the HSI events, including which type of interrupt to use. The choices are data available, fourth entry or FIFO buffer full. Also decide whether to use the peripheral transaction server (PTS) to process the interrupts.
- Initialize the HSI by writing to the correct registers and making sure to be in the correct horizontal window.
- If using the PTS, write the correct values to the PTS control block.
- Initialize the serial port if you are planning to use it to transmit the HSI results.
- Write the appropriate code into your interrupt service routine(s). Make sure that you write the interrupt locations into their correct vector locations.
- Put code into proper areas of memory, starting the main code at 2080H.
- Run the code.

For more information about the HSI, refer to the *8XC196KC/8XC196KD User's Manual* (order # 272238) or the *Ap*BUILDER software (order #272216).

Running the 8XC196KD Target Board...PC-Free

Sean Baartmans
Applications Support
Intel Corporation
Article ID# 1006

The target board that comes with the Project Builder evaluation kit for the 8XC196KD 16-bit microcontroller normally executes code downloaded from a PC. In this case, commands from the PC are used to start executing the code. But the same target board can also be used in standalone fashion. For that, you simply load your application code to an SRAM or EPROM on the board, change one jumper setting, reset the board, and execute your own code.

Normally, the board executes monitor code, called RISM—burned into the one-time-programmable ROM on the 8XC196KD—which interfaces the ECM96 monitor on a PC. Upon reset, the controller looks to memory location 2018H for a chip configuration byte. In the standard mode, location 2018H is in the internal memory and the RISM code begins executing internally at 2080H. When the user gives a "go" command, the RISM jumps to A000H, which is the start of the user's code.

In the standalone mode, the RISM monitor resides in memory locations 2000H to 9FFFH. Thus, the difference between the standard and standalone modes is whether the code from 2000H to 9FFFH is located internally or externally (Figure 1).

To run in standard mode, the code is run from the internal memory. In that case, pin EA# is connected to V_{cc} by leaving the jumper E1 on the board unconnected. In standalone mode, the code is run externally. For that, you must disable the on-board RISM by connecting jumper E1 to AB, in effect grounding EA#. In doing this, you locate the chip configuration byte at 2018H and your code at 2080H externally (Figure 2).

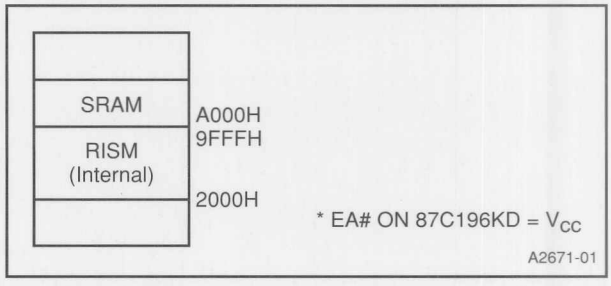


Figure 1. Internal Execution Memory Map

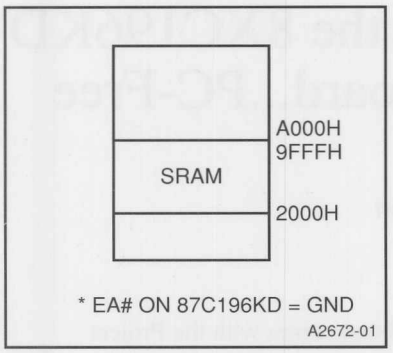


Figure 2. External Execution Memory Map

Memory Matters

Normally, the microcontroller can address up to 64 Kbytes of memory using its 16 address lines (Figure 3). Because an 8-Kbyte SRAM (shipped with the demo board) uses 13 address lines, A0 to A12, loading code at address 2000H is the same as loading it at 0C000H or 0A000H (Figure 4). The standalone mode exploits this feature as the following examples show. Also, Table 1 summarizes the possible standalone modes with different memory sizes.

Because execution starts at 2080H, that's the place to load your code if you want to run in standalone mode. This can be tricky. The RISM code for the target board sits between 2080H and 9FFFH. Yet we must locate the code in external memory at the same locations while internal memory is being used; that is, when pin EA# is high (E1 unjumpered). Because this same memory space is used for the on-board RISM, the problem is how to locate code at 2018H, 2080H and above in external memory. If we switch the EA# pin to force this region to external memory, we can no longer communicate with the PC because we have disabled the RISM located in the on-board ROM.

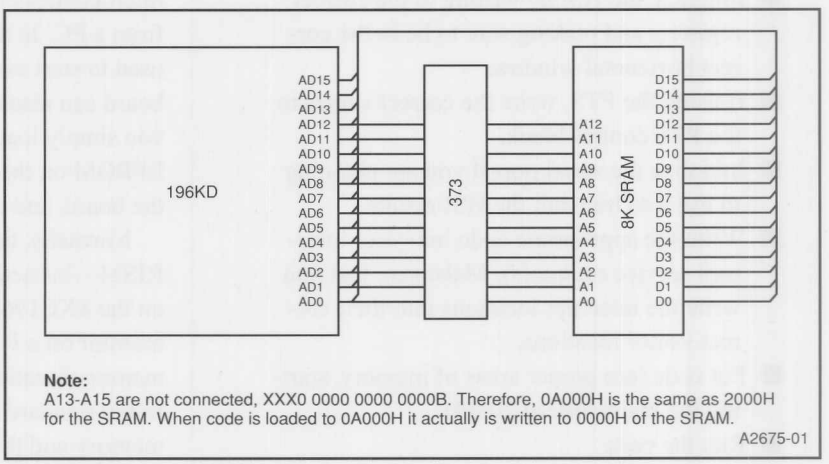


Figure 4. 8XC196 Connections to External 8-Kbyte SRAM

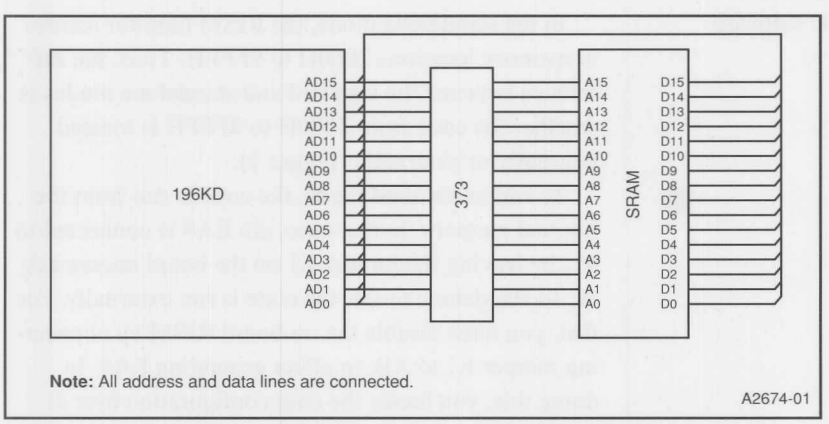


Figure 3. 8XC196 Connections to External SRAM

The solution is to "trick" the memory into thinking that the controller is addressing locations 2018H, 2080H and above when loading code to the external SRAM. Because the internal memory ends at 9FFFH, and only 13 address lines are used with an 8-Kbyte SRAM, address A000H looks like 0000H to the external SRAM.

You simply put the chip configuration byte at A018H and start your code at A080H. This locates the code at 0080H on the SRAM. As

mentioned, because only 13 address lines are used, 2080H looks like 0080H to the SRAM. During reset with EA# pin low (E1 tied to AB), the microcontroller will fetch from 2018H. Again, because the 8-Kbyte SRAM has only 13 address lines, address 2018H looks to it like 0018H. Consequently, your application code starts executing at 2080H, which is 0080H on the SRAM. So after downloading your code to A000H, disconnect from the PC, change E1 to AB, and press reset.

When using a 16- or 32-Kbyte SRAM, the same technique applies (Figures 5 and 6). Here the SRAM has only 14 address lines, so address A000H appears the same as 2000H. When the controller tries to run the program at locations A018H and A080H, these appear as 2018H and 2080H, respectively, on the address bus. You simply download the user code to A018H and A080H, install the jumper on the EA# pin, and press reset.

Table 1. Standalone Mode and Different Memory

Device	Program with	Start code at
8-KB SRAM	Target Board	A080H
16-KB SRAM	Target Board	A080H
32-KB SRAM	Target Board	A080H
64-KB SRAM	(not possible in standalone mode)	
EPROM	Programmer	2080H

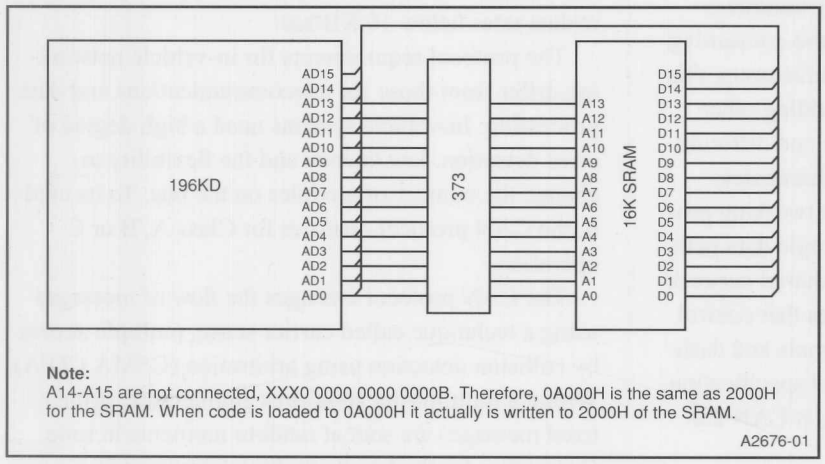


Figure 5. 8XC196 Connections to 16-Kbyte SRAM

Easier with EPROM

Using EPROM in standalone mode is much easier than using SRAM. With EPROM you must burn the code into the exact locations, put the chip configuration register at 2018H, and start your code at 2080H. Then place the EPROM into the socket, make sure that the jumpers are correct for your type of EPROM, jumper the EA# pin for external execution, and press reset, which causes the controller to execute code from external memory.

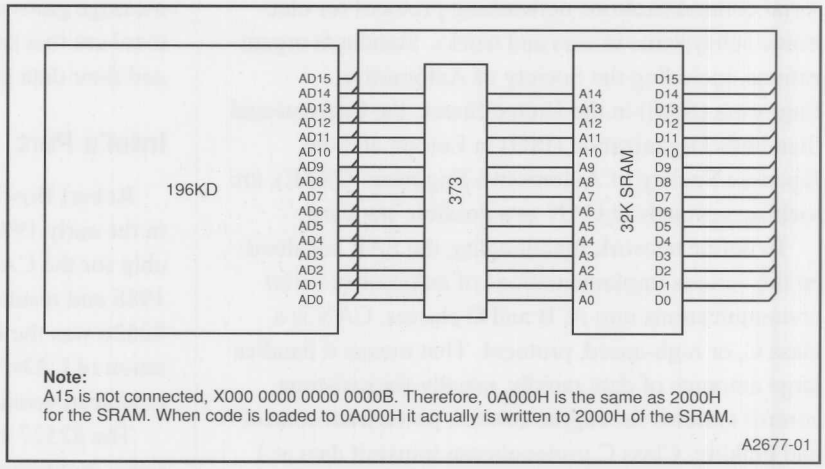


Figure 6. 8XC196 Connections to 32-Kbyte SRAM

CAN, the In-Vehicle Protocol That Could

Article ID# 1007

Circuits, sensors and other electronic components in a typical car or truck number in the hundreds and are on the rise. But in many cases, individual electronic subsystems are separately wired to a control module, giving rise to wire harnesses that are heavy, bulky and difficult to design and install. Because each harness uses its own connectors, the potential for failures is high, and pinpointing these failures is time-consuming if not impossible. To the dismay of manufacturers, this system of point-to-point wiring makes adding other electronic features to vehicles expensive and difficult.

To skirt the need for multiple wiring harnesses, designers have developed multiplexed networking protocols to move electronic data along a single data path, or bus. Instead of separate harnesses, a shared network carries information to individual modules that control functions like anti-lock braking, turn signals and dashboard displays. The pre-eminent protocol specification, and prime candidate for standardization, is CAN 2.0.

CAN Can

CAN, or Controller Area Network, is a high-speed serial communications networking protocol for electronic subsystems in cars and trucks. Standards organizations, including the Society of Automotive Engineers (SAE) in the United States, the International Standards Organization (ISO) in Europe and the Japanese Society of Automotive Engineers (JSAE), are looking seriously at CAN as a possible standard.

To define network functionality, the SAE has divided the various implementations of automotive serial communications into A, B and C classes. CAN is a class C, or high-speed, protocol. That means it handles large amounts of data rapidly, usually for real-time control systems for engine control, powertrain control and braking. Class C protocols can transmit data at 1 Mbit/second, transferring data almost instantaneously.

Class B networks are intended for data communications throughout the vehicle. They carry data, such as engine temperature, that is generated in electronic subsystems and must be passed to other subsystems. This data doesn't control the subsystems and is not transmitted in real time. Data speeds typically range between 10 and 40 KB/sec.

The Class B protocol development has received much attention because of a decision by the California Air Resources Board (CARB) to mandate on-board diagnostics for emission control testing. In this way, state testing centers will be able to plug their computers into a vehicle to check its emissions equipment.

Class A protocols handle low-priority applications operated by the driver's command, such as headlights and electric windows. This protocol typically operates at data rates below 10 KB/sec.

The protocol requirements for in-vehicle networking differ from those for telecommunications and data processing. In-vehicle systems need a high degree of error detection, low latency and the flexibility to change the number of modules on the bus. To its credit, the CAN protocol qualifies for Class A, B or C operation.

The CAN protocol manages the flow of messages using a technique called carrier sense, multiple access by collision detection using arbitration (CSMA CD/A). Within a multimaster serial-bus architecture, prioritized messages are sent at random moments in time. Contention for the bus by multiple messages is resolved by arbitration, where a message identifier travels along the bus bit by bit until the highest priority message gains control. Advantages of this type of protocol are that several subsystems can report at one time and their data processed continuously.

Intel's Part

Robert Bosch GmbH developed the CAN protocol in the early 1980s. Intel worked with Bosch on the first chip for the CAN protocol, the 82526. Introduced in 1988 and reaching full production status in 1989, the 82526 was the first high-speed, production implementation of CAN. Since then, Intel has released an enhanced version, the 82527.

The 82527 silicon handles all communication functions, including message transmission, reception, and error detection and confinement. In this way, it frees the host microcontroller to handle other functions, like engine control or anti-lock braking. This feature is an important attribute of the 82527 and differentiates it from more primitive CAN devices that leave communication management to the host microcontroller.

Fabricated with Intel's CHMOS III technology for maximum performance and integration, the 82527 operates over the automotive temperature range of -40

to +125 degrees Celsius. Other CAN devices are also available from Intel as well as other venders. Future devices are expected to reduce the cost of building systems that use the CAN protocol.

Where CAN Stands

European cars with CAN in-vehicle networks have been produced since 1991. In Europe, the ISO has adopted CAN as the preferred high-speed networking protocol for vehicles. European automakers say they need the CAN protocol now, particularly for luxury models that have highly distributed electronic subsystems. For example, where U.S. automakers often integrate engine and transmission functions into a single module, European automakers have separate injection, ignition and transmission modules.

In the United States, major car and heavy truck makers have also evaluated CAN. Truck manufacturers are especially interested because of the distributed nature of the truck subsystems. The SAE Truck and Bus Control and Communications Network subcommittee selected the CAN protocol as the foundation for J1939, the Class C network for truck and bus applications. The subcommittee voted to adopt CAN because of its:

- Ability to handle high-speed data and increase subsystem reliability
- Ability to send data fast enough for future needs
- Cost-effective implementation and availability in silicon (like the 82527)
- Synergy with the work of other standards organizations

With the SAE Truck and Bus subcommittee's decision to use CAN, the protocol is being designed into U.S.-built trucks and buses. As a result, the hundreds of wires between subsystems have been reduced to a few, with the effect of speeding manufacturing, simplifying diagnostics and enhancing the capacity to add options. Many of these designs will reach production and be on the road in the 1994 model year.

Both the JSAE and the SAE Class C Task Force have evaluated CAN and are likely to adopt it as a standard. In addition to the automotive market, the CAN protocol is being designed into non-automotive networking products like factory automation equipment, elevators, medical equipment and in-home electronic communications systems.

Automakers and Intel on ABS: "Full-Speed Ahead!"

Article ID# 1008

One of the fastest growing markets in automotive electronics is for the microcontrollers used in anti-lock braking systems. The overall value of microcontrollers in North American automotive vehicle-control applications, of which ABS is a significant portion, will grow from about $70 in 1993 to $190 by the year 2000, according to BIS Strategic Decisions of Luton, England.

Several factors are contributing to this growth. For one, U.S. and foreign governments are encouraging auto manufacturers to make cars safer. Consumers also are demanding more safety-related features as these become increasingly important to them. As a result, more than 40 percent of passenger cars and light-duty trucks built in North America have ABS, according to Tier One, an automotive industry research firm in Mountain View, California. By 2000, predicts Tier One, this figure will approach 100%. Projections indicate similar growth for Japan and Europe.

Finally, the cost of anti-lock braking systems (ABS) is dropping. Intel, for example, expects that the prices to car makers for four-wheel ABS will fall below $150, down from the current range of $250 to $400.

What is ABS?

ABS prevents a vehicle's wheels from locking when its brakes are applied suddenly. Such locking causes a car skid. ABS works by modulating, or pulsing, the brakes when the driver applies them hard. This automatic pulsing improves stopping distances and gives better steering control. As a result, it gives a driver greater control over the car during sudden braking.

Advanced systems combine anti-lock braking with traction control. By adjusting the limited-slip differential or throttle control during acceleration, traction control helps the vehicle steer better and gain traction, especially on wet or slick surfaces.

While combined ABS/traction control (ABS/TC) systems are more expensive and less in demand than ABS alone, manufacturers expect to equip increasing numbers of car models with these combined systems as costs drop and consumers become more aware of their

Continued from page 17

benefits. Robert Bosch GmbH estimates that by the year 2000, 20% of the total ABS business will be for ABS/TC systems.

The Elements of ABS

The three key electronic elements of ABS are wheel speed sensors; brake modulators; and the electronic control unit (ECU), which is the system's electronic brain. A microcontroller, like those Intel supplies to ABS module manufacturers, acts as the central processing unit. Residing within the ECU, the microcontroller monitors wheel speed and road conditions every five to seven milliseconds, processing this information as it gets it from speed sensors connected to a vehicle's wheels.

In adverse driving condition, the microcontroller controls the modulators which, in turn, adjust the pressure applied to the brakes. By varying that pressure, and in particular reducing it if the wheels begin to lock, the system prevents the tires from skidding.

Today, most ABS-equipped passenger cars have four-wheel systems, in which sensors collect data from—and the system controls—all four wheels. Most light-duty trucks with ABS have two-wheel systems, which control only two of the four wheels. Forecasts suggest, however, that by the middle of this decade virtually all systems will be four-wheel types, offering superior vehicle handling and more control than two-wheel types.

ABS and traction control systems use either 8- or a 16-bit microcontrollers to process data. The main difference is the amount of data processed simultaneously, with a typical 16-bit device processing data up to 10 times faster than its 8-bit counterpart. This means a 16-bit microcontroller can handle more data faster and more accurately. Thus, a 16-bit device not only operates more quickly, yields greater braking resolution and improves performance, it can even govern a traction control system at the same time.

Today, many systems use 8-bit devices. However most are moving to the higher processing speeds, higher integration and improved performance of newer 16-bit controllers. As the cost of 16-bit devices continues to decrease, they are becoming affordable for even economical ABS designs.

Intel's Role

Intel's Automotive Operation is the largest supplier of microcontrollers for passenger car and light-duty truck ABS and traction control systems. The first integrated ABS, introduced by ITT Automotive (formerly Alfred Teves GmbH) in 1983, used an Intel 8051 8-bit microcontroller. A year later, Robert Bosch GmbH introduced a system using an Intel 16-bit microcontroller.

Today, more than 75 percent of all new 16-bit ABS and ABS/TC designs in North America and Europe use Intel microcontrollers. In all, Intel has captured more than 40 percent of the market volume in the passenger cars. Among the leading ABS and traction control systems suppliers that use Intel controllers are ITT Automotive, Bosch, Kelsey-Hayes, Bendix, Knorr-Bremse, Wabco and Lucas Girling. In turn, these suppliers provide modules to car manufacturers including Ford, General Motors, Chrysler, BMW, Mercedes Benz, Volkswagon, Saab, Porsche, Audi, Renault, Nissan and Volvo.

What Designers Demand

The performance features that ABS designers need in 16-bit controllers such as Intel's MCS® 96 family include a watchdog timer, on-chip memory for software control and diagnostic functions, a dedicated hardware interrupt controller and I/O manager for real-time event control, and an A/D converter.

Besides performance and price, module designers look for architectural "headroom" when selecting a microcontroller. That is, they prefer a microcontroller with several variations and the capacity to provide expanded system performance. Toward this end, by choosing a family of microcontrollers that shares the same architecture, designers can use different models in various ABS modules without costly and time-consuming redesigns and recertifications. Software in particular takes a large investment. Because manufacturers put much time and money into developing fail-safe systems, compatibility between software and microcontroller architecture is of prime importance.

Intel's newest 16-bit controllers provide that compatibility across products and, by offering numerous combinations of features and processing capabilities, expand a module's designs. A module supplier can, for example, design an ABS incorporating the simplest member of the family for cost-sensitive design and grow to one incorporating the most complex module to meet higher system needs such as those required by ABS/TC systems.

And because the architecture in each case is the same, the software can be reused. In this way, the module supplier saves time and money, while reducing its risk, by not having to rewrite expensive software for each kind of ABS module. In addition, because the company's designers need be familiar with only one basic architecture, the company benefits from shorter learning curves and a shorter time-to-market.

ABS and Beyond

Four-wheel ABS is expected to be a standard feature on virtually all passenger cars and light-duty trucks by the end of the decade. Because of the rapid commoditization of this market, system and component suppliers will push aggressively for rapid cost reductions.

Over the longer term, suppliers will focus on how ABS and ABS/TC will evolve. Most industry observers and participants feel that this evolution is toward vehicle dynamics control (VDC). Going well beyond the capacity of ABS/TC systems, VDC systems control the pitch, roll and yaw of a vehicle in any handling situation.

For example, imagine a car, traveling too fast as it enters a sharp corner covered by black ice. A VDC system will control engine, transmission, braking, suspension and steering responses to help the driver maneuver the vehicle safely. Robert Bosch, a leader in this technology, predicts that VDC systems will account for roughly 5 percent of the total vehicle control (ABS, ABS/TC and VDC) market by the year 2000.

Intel not only offers the most comprehensive 16-bit product line in the industry for ABS and ABS/TC, it has products well-suited for the emerging VDC market as well. As these and other systems evolve, Intel's automotive operation will continue to provide a broad range of standard and application specific logic and memory products for the automotive and heavy-vehicle industry.

Intel Automotive History

1979: Intel focuses on automotive electronics as five car makers and suppliers approach the company to develop an electronic engine controller (EEC). This controller will become the "grandfather" of the 8096.

1980 to 1981: Ford and Intel strike EEC deal. Dialogue begins with a leading anti-lock braking system (ABS) manufacturer regarding an 8-bit controller for ABS. Auto activity is managed through Intel's Consumer Operation.

1982: Intel starts development of the 8096AH.

1983: The 8061 EEC reaches production, and volume shipments begin. The first ABS based on Intel's 8-bit 8051 microcontroller is introduced and becomes the seed of Intel's success with ABS. A leading European car supplier begins using the 8048 and 8049 for engine control.

1984: Leading European ABS supplier approaches Intel for development of a 16-bit ABS controller, which becomes the 8X9X. Dialogue with Japan begins. EEC production reaches high volume.

1986: Intel forms Automotive Operation. It enters a joint venture with Bosch to produce the Controller Area Network (CAN) protocol. American manufacturers start to use the 8051 to control such dashboard functions as instrumentation, clocks and entertainment equipment.

1987: Intel's Automotive Operation begins three-prong application focus on engines, networks and brakes. ABS and engine controller volumes ramp sharply. Development of 16-bit CMOS ABS controller begins. Both 8- and 16-bit devices gain widespread design wins.

1988: 16-bit CMOS controller for ABS hits production volume. Intel wins design at another North American ABS supplier.

1989: Intel ABS customer portfolio now includes four of the top five ABS suppliers using either 8- or 16-bit Intel architectures. The 82526 CAN controller reaches production status.

1990: 8XC196KR family launch targeted at the market for brakes. Four of five top ABS suppliers now design 16-bit systems.

1991: ABS controller business becomes top revenue source. Bosch-Intel Design Group (BiDG) is formed. Intel wins supplier quality awards from three automobile makers or suppliers. First production car with CAN is introduced.

Revenue History

	Automotive Operation ($M)	Intel Corporation ($B)
1989	115	3.13
1990	122	3.92
1991	127	4.78
1992	140	5.84
1993	155	8.78

Manufacturing Locations

Wafer Fabrication

- Fab 4, Aloha, Oregon
- Fab 6, Chandler, Arizona
- Fab 7, Rio Rancho, New Mexico

Assembly Plants

- A01, Penang, Malaysia
- A03, (ANAM). Seoul, South Korea
- A02, Manila, Philippines

Test Facilities

- T03, Penang, Malaysia

1992: The 82527 CAN+ is introduced in August. Intel Automotive wins flash-memory design-in at U.S. auto maker. The 87C196KT, an enhanced ABS controller, becomes available.

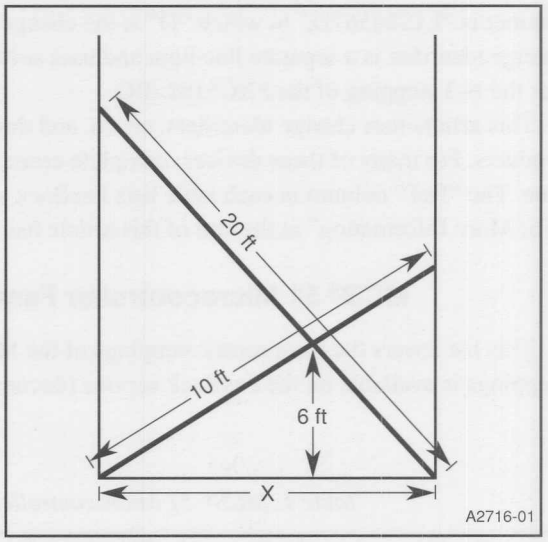
1993: Intel achieves two major shipment milestones by delivering 20 million ABS and 40 million engine-control microcontrollers. Also, the 87C196JT, a low-cost, highly integrated ABS controller is introduced.

New Embedded Control FaxBack* Service Documents

New/Updated as of August, 1994

Title	Number
56L TSOP Programming Adapters: E28F016SV, E28F016SA & DD28F016SA Flash File™ Memories	2226
56L SSOP Programming Adapters: DA28F016SV, DA28F016SA Flash File™ Memories	2259
Flash: 2/4/8-Mbit Boot Block Family Overview	2263
8XC196NT: User's Manual Errata	2338
8XC186/C188: Flash Memory Interface Recommendations	2342
Altera "Dear Customer" letter to Intel PLD/FPGA Customers	2354
Article: MCS® 96: Intel's Automotive Operation and the ABS/Traction Control Market	2356
Article: CAN: Intel's Automotive Operation and In-Vehicle Networking	2357
8XC196KR/JR/KQ/JQ: 1B00H-1BDFH Bug	2358
MCS® 96 processor: C 196 LOG, LOG10, POW and EXP routines	2361
Embedded i386™ Processor: Hardware and Software Development Tools Line Card	2765

After Hours Activity



A 10- and a 20-ft. ladder cross an alley and lean against the buildings as shown. They intersect at a height of 6 ft. Without using graphical means, calculate the width of the alley.

(Hint: The alley is less than 6 ft. wide.)

Article ID #1009

Change identifiers have been used since 1990 to distinguish revisions, or steppings, of embedded control devices. Older devices have no change identifiers. On most devices, the change identifier is the last character in the FPO number, which is typically a nine-character code on the second line on the top of the device. An example FPO number is "L1234567D," in which "D" is the change identifier. On some devices, such as the 8XC51SL-BG, the change identifier is a separate line item and uses several characters. For example, change identifier "SW011" identifies the B-3 stepping of the 8XC51SL-BG.

This article lists change identifiers, errata, and design considerations for recent steppings of embedded control products. For many of these devices, complete errata listings or explanations are available from the FaxBack* service. The "Ref" column in each table lists FaxBack service document numbers for errata lists and explanations, and "For More Information" at the end of this article has a complete list of related document numbers and titles.

MCS® 51 Microcontroller Family Errata and Design Considerations

This list covers the most recent steppings of the MCS® 51 microcontrollers. A complete list of the errata for all steppings is available on the FaxBack service (document #2632).

Table 1. MCS® 51 Microcontroller Family Errata and Design Considerations

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8051AH/8031AH	C	A	1. External interrupt 0 errata	2154 2161
	C-3	B	No known errata	
80C51BH/80C31BH	C	none	1. Reset lockup problem 2. High IPD if C does not equal B.7 before going into powerdown 3. ROM verify mode fails 4. Steam passivation problem on plastic parts	
	C-1	none	1. High IPD if C does not equal B.7 before going into powerdown 2. ROM verify mode fails	
	D	D or 2	No known errata	
87C51	D	A	No known errata	2106
80C52/80C32	C	none	1. RST/ONCE mode problem	
	A	A	No known errata	
83C51FA/80C51FA	C	none	1. PCA errata	2528
87C51FA	C	none	1. RST/ONCE mode problem 2. PCA errata	2528
	D	A	No known errata	2107
8XC51FB	A	none	1. PCA errata	2528
	B	A	No known errata	2111

Table 1. MCS® 51 Microcontroller Family Errata and Design Considerations (Continued)

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8XC51FC	A	none	1. Port 1, 2, 3 problem — asychronous port reset not supported 2. Failed ESD qual testing	2028
	B	none	No known errata	
8XC51GB	B	none	1. Reset polarity changed to active low 2. Port 1 reset value changed to all zeros	2032 2032
	B-2	none	No known errata	
8XC152JX	B	none	1. AE/RDN race condition 2. Receive FIFO is not cleared when receiver is enabled 3. DMA errata 4. SDLC flag recognition errata	 2030 2118 2035
	C	none	No known errata	
8XC51SL-BG	B-3	SW011	1. GATEA20, RCL hardware speedup processing 2. Powerdown current stabilization 3. Port 2 address mux 4. KSI powerdown wakeup interrupt 5. Reset errata	2008 2114
	B-4	SW062	1. GATEA20, RCL hardware speedup processing 2. Powerdown current stabilization 3. Port 2 address mux 4. KSI powerdown wakeup interrupt	2008
8XC51SL-AH/AL	A-0 A-1 A-2	AA BA CA	1. Power-down current stabilization 2. System power management errata	2048

MCS® 96 Microcontroller Family Errata and Design Considerations

This list covers the most recent steppings of the MCS® 96 microcontroller. Complete lists of the errata for all steppings are available on the FaxBack service for the 8X9XBH (#2134), the 8XC196KB (#2548), the 8XC196KC (#2136), the 8X196KD (#2315), the 8XC196KR (#2527), and the 8XC196NT (#2178).

Table 2. MCS® 96 Microcontroller Family Errata and Design Considerations

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8X9XBH	D	D	1. Indexed 3-operand multiply 2. HSI FIFO 3. Reserved location 2019H 4. RESET and the QBD pins 5. Software RESET timing 6. Using T2CLK for Timer2	2134
	E	E	1. Indexed 3-operand multiply 2. HSI resolution 3. Reserved location 2019H 4. Reserved location 201CH	 2631 2140

Table 2. MCS® 96 Microcontroller Family Errata and Design Considerations (Continued)

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8XC196KB 8XC196KB10/KB12	All		1. HSI 8/9 State 2. CMPL with R0	2548
	B	B	1. Divide during HOLD or READY 2. SIO framing error 3. SIO RI flag 4. DJNZW instruction 5. ALE glitch 6. HSI_MODE divide-by-eight	 2568 2192
8XC196KB/KB16	B	B, D	1. Divide during HOLD or READY 4. Missed EXTINT P0.7 3. HSI_MODE divide-by-eight 4. Oscillator sensitivity	2122 2049 2192
	C-0 C-1	E F, G	1. Missed EXTINT P0.7 2. HSI_MODE divide-by-eight	2049 2192
8XC196KC			The number following each entry in this list is a cross-reference to the applicable section of FaxBack service document #2136.	2136
	all		Design Considerations 1. Indirect shift count value 116 2. Write cycle during Reset 147	
	B-1	B	1. Divide error during hold 109 2. NMI during PTS skips address 123 3. QBD glitch during powerup 163 4. ONCE mode entry 214 5. Oscillator startup 215 6. Reset hysteresis 216 7. Missed EXTINT P0.7 8. HSI_MODE divide-by-eight 9. Aborted PTS skips next byte	 2049 2192 2328
	B-3	D or E	1. Divide error during hold 109 2. NMI during PTS skips address 123 3. QBD glitch during powerup 163 4. Reset hysteresis 216 5. Missed EXTINT P0.7 6. HSI_MODE divide-by-eight 7. Aborted PTS skips next byte	 2049 2192 2328
	D	H,J,L,M	1. Missed EXTINT P0.7 (80C196KC only) 2. HSI_MODE divide-by-eight. 3. IPD Hump	2049 2192 2311
8XC196KD	A-1	B	1. Missed EXTINT P0.7 2. HSI_MODE divide-by-eight 3. IPD Hump	2049 2192 2311
	B	D,E	1. Missed EXTINT P0.7 (83C196KD only) 2. HSI_MODE divide-by-eight 3. IPD Hump	2049 2192 2311
8XC196KR/JR/KQ/JQ	A, C	A, C	Design Considerations 1. P6_REG not updated immediately	2527

Table 2. MCS® 96 Microcontroller Family Errata and Design Considerations (Continued)

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
8XC196KR/JR/KQ/JQ (Continued)			2. Write cycle during reset 3. EPA timer reset/write conflict 4. Valid time matches 5. CLKOUT 6. Indirect shift operation 7. Internal RAM powerdown leakage 8. A/D latchup 9. INST operation 10. KQ/JQ memory map	
	A	A	**Errata** 1. Oscillator noise sensitivity 2. Slave programming mode 3. A/D abort 4. PTS with other interrupts 5. PTS/NMI conflict 6. Data output register cleared when mode register is written 7. Divide error during Hold/Ready 8. SIO Mode 0 9. Remap mode on EPA3 10. Serial port framing error 11. EPAIPV value multiplied by 2 12. EPA_MASK1/EPA_PEND1 must be written as words 13. Interruptable block move (BMOVI)	
	A, C	A, C	1. Ioh2 = – 6 µA	
	C,D	C,D	1. Cannot access external locations 1B00H-1BDFH	
8XC196NT	D	D	Design Considerations In bus controller modes 1 and 2, in 8-bit bus mode, the upper address lines need to be latched 1. Illegal Opcode interrupt vector not taken. 2. Aborted interrupt vectors to lowest priority. 3. PTS request during interrupt latency.	
8XC196MC/MD	B	B	No known errata	
8XC196NP	A	A	1. Illegal Opcode interrupt vector not taken. 2. Aborted interrupt vectors to lowest priority. 3. PTS request during interrupt latency.	

8XC186/8XC188 Family Errata and Design Considerations

Table 3. 8XC186/8XC188 Family Errata and Design Considerations

Device	Step	Change Identifier	Errata and Design Considerations	Ref.
80C186A, B	none		1. Non-contiguous Interrupt Acknowledge cycles 2. ERROR# processing during FWAIT instructions 3. Input high voltage requirement on SRDY and ARDY pins 4. Interrupt Status Register (DHLT and Timer Interrupts) 5. Bus preemption errata (HOLD/HLDA protocol) 6. 80C188 RFSH# pin output timing	2096
80C186XL	A	none	Never put into production	
	B	A	1. INTx/INTAx in Cascade Mode	2025
	C	B	No known errata	
80C186EA/80L186EA	A	A	1. Low hysteresis on RESIN# pin 2. TEST/BUSY#, RD#/QSMD#, LCS#, and UCS# input low voltage 3. INTx/INTAx in Cascade Mode	2025
	B	B	1. INTx/INTAx in Cascade Mode	2025
80C186EB/80L186EB	A	A or none	1. Entry into ONCE mode 2. Low hysteresis on RESIN# pin 3. SINT1 input not latched internally 4. Ready input during INTA# bus cycle 5. CLKOUT transitions on the rising of CLKIN instead of the falling edge 6. I/O ports 1 and 2 initialize to Port instead of Peripheral function (documentation error) 7. INTx/INTAx in Cascade mode	2025
	B	B	1. INTx/INTAx in Cascade Mode	2025
80C186EC	A	A	1. Early exit from Reset (with high Vcc, or low temperature, or both) 2. Powersave Mode initialization at Reset	
	B	B	No known errata	

For More Information

The following FaxBack* service documents contain errata lists and explanations.

Title	Number
MCS® 51 Controller Errata	
MCS 51: Errata and Design Considerations	2632
MCS 51: 1991 8 bit Microcontroller Book Errata	2123
MCS 51: 1992 8-bit Datasheet Errata	2165
MCS 51: FX-Core ALE Disable Datasheet Errata	2308
8XC51FX: QFP Pin 39 Change	2124
8XC51FX: PCA / TIMER2 Errata	2528
8XC51FX: Hardware Description Errata	2017
87C51FA: D Step	2107
87C51FB/83C51FB: B Step	2111
87C51FC: Data Sheet Errata I	2024
87C51FC: Data Sheet Errata II	2020
87C51: D Step	2106
87C54/80C54: Data Sheet Errata I	2022
87C54/80C54: Data Sheet Errata II	2021
8051/31AH: Shrink C-Step External Interrupt 0 Errata	2161
80C31/51BH: D-Step Marking	2015
87C51GB: A-1 & B Errata & Design Consider	2032
8XC51SLAH/AL Errata	2048
80C51SL-BG: Errata Version 2.6	2008
80C51SL-BG: Product Preview Data Sheet Errata	2630
80C51SL-BG: Errata	2130
80C51SL-BG: Reset Errata	2144
80C152: SDLC Flag Recognition Bug	2035
8XC152: External Demand DMA Bug	2129
8XC152: Global Serial Channel Bug	2030
8XC152: DMA Bug	2118
8XC152: Errata and Clarifications III	2043
MCS® 96 Controller Errata	
8XC196KB/KC/KD,8XC198,8XC194: HSI Events (9 or greater)	2052
Project Builder: Manual Errata	2170
8XC196KB: History and Errata	2548
* 80C196KB/83C196KB: B-0 to C-1 Stepping Conversion	2316
8XC196KB: ALE Glitch	2568
8XC196KB/KC/KD: HSI_MODE divide-by-eight	2192
8XC196KB/KC/KD: Missed EXTINT Interrupt Problems on PO.7	2049
8XC196KC: 1991 Handbook Errata	2131
8XC196KC: HSI PTS Handbook Errata	2141

Title	Number
MCS® 96 Controller Errata (Continued)	
8XC196KC: Bug List	2136
8XC196KC/KD: Powerdown Current (IPD) Hump	2311
8XC196KC/KD: 1992 User's Manual Errata	2570
8XC196KD: Bug List	2315
8XC196KR/JR/KQ/JQ: Errata	2527
8XC196KR Eval Board EPA Remap Warning	2120
8XC196: Hold/Ready DIV/DIVB Problem	2122
8XC198 and 8XC196KB/KB16: Data sheet Errata	2569
8X9XBH, 8X9XJF: Bug List	2134
8X9X: Reading 201CH Bug	2140
186 Controller Errata	
80C186: C186/188 Compatibility with 80C186XL / C188XL C-Step	2132
80C186: AMD 80C18x to INTEL 80C186/186XL Comparison	2540
80C18x XL/EA/EB: INTx/INTAx# Errata	2025
EV80C186EB: Eval Board Manual Errata	2158
EV80C186: 186 Family Eval Board Errata	2592
186XL/EA/EB/EC: 186 Family User's Manual Errata	2603
80C186/C188: B Step Technical Bulletin	2100
80C186/C188XL: B-Step Technical Bulletin	2101
80C186/C188EA: A-Step Technical Bulletin	2102
80C186/C188EB: A-Step Technical Bulletin	2103
80C186/C188EB: B-Step Technical Bulletin	2104
80C186/C188EC: A-Step Technical Bulletin	2105
80C18xEC/80L18xEC: B-Step Technical Bulletin	2057
Embedded Intel386™ Microprocessor Errata	
Errata: Intel386 EX A-Step Errata	2762
Document Corrections: Intel386EX Corrections/Additions to EAS Rev. 3.0	2193

intel®

FIRST CLASS MAIL
U.S. POSTAGE
PAID
PHOENIX, AZ
PERMIT No. 3559

Intel Corporation
EMD Applications – CH3-64
5000 W. Chandler Blvd.
Chandler, AZ 85226

READER'S FEEDBACK

Please fax your completed questionnaire to
1-800-772-2862 or 1-602-554-7436

Name: ____________
Title: ____________
Company: ____________
Address: ____________
City: ____________
State/Country: ____________
Zip Code: ____________
Phone: ____________
Fax: ____________
Intel products used: ____________
Description of application: ____________

☐ Add my name to your mailing list.
☐ Delete my name from your mailing list.

We value your opinion!

1) Please rate the quality of this publication:
☐ Poor ☐ Acceptable ☐ Excellent

2) Please rate the usefulness of this publication:
☐ Not Useful ☐ Somewhat Useful ☐ Very Useful

3) Please rate the individual articles:

ID #	Quality: Poor	Quality: Adequate	Quality: Excellent	Usefulness: None	Usefulness: Some	Usefulness: Much
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐
____	☐	☐	☐	☐	☐	☐

4) Please list the quarter/year of this issue:

5) Please list topics you would like to see in future issues: ____________

Embedded Applications Journal
Fourth Quarter, 1994
Managing Editor: Steven M. McIntyre

The Embedded Applications Journal is a quarterly publication of Embedded Applications Support, Embedded Tools Engineering, and Embedded Market Development, who are part of the Embedded Microcontroller Division of Intel Corporation.

Literature Order Number 241294-011